สุทธิศักดิ์ พงศ์ธนาพาณิช

เปิดโลกการเข้ารหัสข้อมูลด้วยภาษาซี ตอน Shannon-Fano Coding

นำเสนอวิธีการเข้ารหัสข้อมูลที่มีประสิทธิภาพเป็นที่รู้จักกันดี

ผู้อ่านที่ติดตามบทความและตำราทางด้านภาษาเบสิกที่ผมเขียนขึ้นมาคงมีความแปลกใจว่าทำไมบทความนี้ผมจึงเสนอการเขียนโปรแกรมโดยใช้ภาษาซี ทั้งนี้ก็เพราะการเขียนโปรแกรมเกี่ยวกับการบีบอัดข้อมูล (data compression) เป็นการเขียนโปรแกรมที่มีการประมวลผลข้อมูลสตริงระดับบิต ซึ่งการทำงานด้านนี้ไม่เหมาะกับภาษาเบสิก เนื่องจากตัวแปลภาษาเบสิกไม่ได้ถูกออกแบบเพื่อทำงานระดับบิต แต่ทั้งนี้ไม่ใช่ว่าเราไม่สามารถเขียนโปรแกรมการบีบอัดข้อมูลด้วยภาษาเบสิก แต่สำหรับภาษาซีเนื่องจากตัวแปลภาษาได้มีคำสั่งและโครงสร้างของภาษาที่อำนวยให้สามารถประมวลผลข้อมูลประเภทนี้ได้อย่างรวดเร็ว และที่สำคัญภาษาซีเป็นภาษาที่มีความยืดหยุ่นสูงสามารถรันได้บนหลาย ๆ แพลตฟอร์ม

สำหรับผู้ที่สนใจต้องการรู้การทำงานของโปรแกรมบีบอัดข้อมูลที่มีชื่อเสียงในปัจจุบัน เช่น Pkzip หรือ Lharc ก็ขอให้ติดตามบทความนี้ต่อไปนะครับ เพราะผมจะมีอัลกอริทึมและโปรแกรมตัวอย่างแสดงการทำงานที่เทียบเคียงกันได้ สำหรับโปรแกรมตัวอย่างทั้งหมดในบทความนี้ผมได้ทำการทดสอบบน MSDOS เวอร์ชัน 5.0 และใช้ตัวแปลภาษาของบริษัทไมโครซอฟต์ ดังนี้

- QuickC เวอร์ชัน 2.0
- Microsoft C เวอร์ชัน 6.0

ดังนั้นถ้าหากผู้อ่านพบว่าส่วนหนึ่งส่วนใดของโปรแกรมทำงานไม่ถูกต้องหรือเมื่อทดสอบบนตัวคอมพายเลอร์อื่น ๆ เช่น TurboC ได้ผลประการใดกรุณาแจ้งผลการทำงานกลับมายังกองบรรณาธิการวารสารไมโครคอมพิวเตอร์ เพื่อประโยชน์ในการปรับปรุงโปรแกรมในโอกาสต่อไป

อาณาจักรของการบีบอัดข้อมูล

เทคนิคการบีบอัดข้อมูลในปัจจุบันได้ถูกแบ่งออกเป็น 2 ตระกูลคือ

Lossy data compression เป็นการบีบอัดข้อมูลที่สามารถให้ความถูกต้องของข้อมูลสูญหายได้บ้าง ทั้งนี้ก็เพื่อเพิ่มความเร็วของการบีบอัดข้อมูล ซึ่งมันได้ถูกนำไปใช้กับงานด้านกราฟิกอิมเมจ ซึ่งเทคนิคด้านนี้สามารถปรับให้มีความถูกต้องได้หลายระดับ แต่ถ้าหากถูกกำหนดให้มีความแม่นยำสูงก็จะทำให้ขนาดไฟล์ที่บีบอัดข้อมูลมีขนาดใหญ่ขึ้นและใช้เวลาในการประมวลผลนานขึ้นเช่นกัน ซึ่งในปัจจุบันนี้ได้มีฮาร์ดแวร์หลายประเภทที่สามารถบีบอัดข้อมูลโดยใช้เทคนิคนี้

Lossless data compression ไฟล์ที่ถูกบีบอัดข้อมูลโดยเทคนิคนี้จะต้องมีความถูกต้องของข้อมูลเช่นเดียวกับข้อมูลต้นฉบับทุกประการ ซึ่งโดยส่วนใหญ่จะถูกนำไปประยุกต์ใช้กับไฟล์ประเภทฐานข้อมูล สเปรดชีต เวิร์ดโปรเซสเซอร์ หรือใช้บีบอัดข้อมูลที่ส่งตามเครื่องแฟกซ์เป็นต้น ซึ่งข้อมูลที่เก็บในไฟล์ประเภทนี้ล้วนมีความสำคัญทุก ๆ ไบต์ ดังนั้นถ้าหากมีการสูญหายหรือเปลี่ยนแปลงข้อมูลในไฟล์บีบอัดขนาดเพียงบิตเดียว ก็อาจจะทำให้ข้อมูลของไฟล์ที่ถูกขยายขนาดแล้วมีความคลาดเคลื่อนไปทั้งหมดก็ได้

โปรแกรมบีบอัดข้อมูลที่ควรรู้จัก

โปรแกรมบีบอัดข้อมูลทั่วไปเริ่มมีมากกว่า 10 ปีมาแล้ว โปรแกรม COMPACT ซึ่งทำงานบนระบบปฏิบัติการยูนิกซ์เป็นโปรแกรมที่มีการทำงานบีบอัดที่ลำดับ 0 (order-0 compression) โดยใช้เทคนิค adaptive Huffman coding ซึ่งผลลัพธ์ที่ได้ดีพอสมควร แต่เป็นโปรแกรมที่มีการทำงานช้ามาก

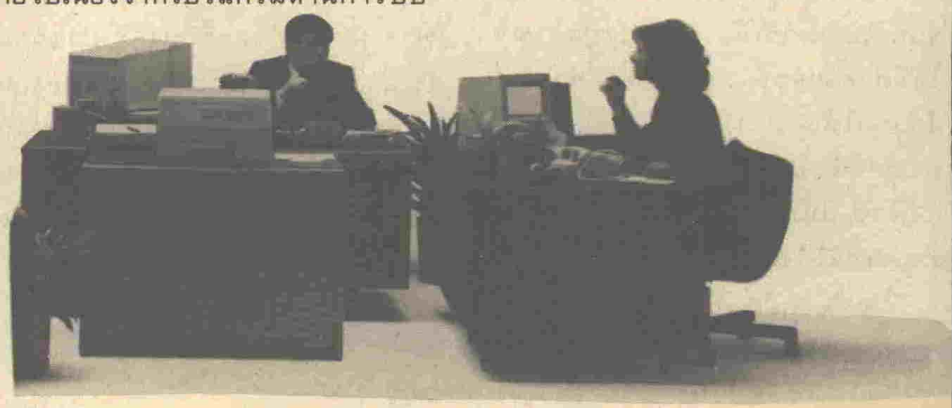
COMPRESS เป็นอีกโปรแกรมที่ทำงานบนระบบปฏิบัติการยูนิกซ์ ซึ่งทำการบีบอัดข้อมูลโดยใช้วิธี LZW dictionary-based compression เป็นโปรแกรมที่ทำงานได้ดีกว่าโปรแกรม COMPACT และสามารถทำงานได้เร็วกว่าอีกด้วย

ในอดีตที่ผ่านมา เครื่องคอมพิวเตอร์แบบตั้งโต๊ะได้ให้การสนับสนุนระบบปฏิบัติการ CP/M และ MS-DOS โดยในเวลาต่อมาก็เกิดโปรแกรมบีบอัดข้อมูล SQ ซึ่งมีการทำงานเทียบเคียงได้กับโปรแกรม COMPACT โดยใช้เทคนิคการบีบอัดข้อมูล static Huffman tree compression โปรแกรมนี้ได้ถูกใช้มากในสมัยนั้นในการสื่อสารระยะไกล

ARC เป็นโปรแกรมบีบอัดข้อมูลต่อมาที่ได้รับความนิยมอย่างมาก เพราะนอกจากสามารถบีบอัดข้อมูลแล้วยังสามารถรวมไฟล์เข้าเป็นไฟล์เดียวได้อีกด้วย ซึ่งโปรแกรมนี้ทำการบีบอัดข้อมูลโดยใช้เทคนิค run-length encoding, order-0 static Huffman coding หรือ LZW compression

แต่ในปัจจุบันนี้ ARC กลับเลือนหายไปเนื่องจากโปรแกรมด้านการบีบอัดข้อมูลในปัจจุบันได้รับการพัฒนาให้มีความสามารถมากขึ้น ขนาดโปรแกรมเล็กและสามารถทำงานได้รวดเร็วขึ้น ที่สำคัญสามารถใช้งานได้ง่ายขึ้น เช่น PKZIP ซึ่งเป็นโปรแกรมที่มีใน share-ware ได้รับการพัฒนาโดย PKWare ซึ่งทำการบีบอัดข้อมูลโดยใช้อัลกอริทึม dictionary-based compression ที่แตกต่างกัน หรือ LHarc ซึ่งเป็นโปรแกรมประเภท freeware เขียนโดย Haruyasu Yoshizaki ชาวอาทิตย์อุทัย ซึ่งทำการบีบอัดข้อมูลโดยใช้เทคนิค LZSS ซึ่งเป็นเทคนิคที่แตกย่อยจากเทคนิค LZ77 compression ตามด้วยขั้นตอน Dynamic Huffman postprocessing หรือ ARJ ซึ่งเป็นโปรแกรมที่ถูกพัฒนาโดย Robert Jung เป็นโปรแกรมที่สามารถบีบอัดข้อมูลได้ดีกว่า LHarc เล็กน้อย

แต่ความก้าวหน้าของการบีบอัดข้อมูลไม่ได้หยุดแค่นี้ เพราะโปรแกรมในปัจจุบันนอกจากจะบีบอัดข้อมูลทีละไฟล์แล้ว ยังสามารถบีบอัดข้อมูลทั้งไดรฟ์ได้อีกด้วย เช่น โปรแกรม STACKER หรือ Double-Space เป็นต้น โปรแกรมประเภทนี้จะทำการบีบอัดไฟล์ทุกครั้งที่มีการจัดเก็บไฟล์ลงบนดิสก์และขยายข้อมูลคืนสู่สภาพเดิมเมื่อมีการอ่านไฟล์สู่หน่วยความจำโดยที่ผู้ใช้ไม่จำเป็นต้องเรียกใช้โปรแกรมแบบ command-line



การบีบอัดข้อมูลคืออะไร

โดยทั่วไปการบีบอัดข้อมูลประกอบด้วยการอ่านสายข้อมูลเข้ามาแล้วทำการแปลงข้อมูลเหล่านี้ให้อยู่ในรูปแบบโค้ดต่าง ๆ ถ้าหากการบีบอัดข้อมูลมีประสิทธิผล โค้ดที่ได้จะต้องมีสั้นกว่าข้อมูลเดิม โดยที่โค้ดที่ได้จะเป็นอย่างไรนั้นก็ขึ้นกับโมเดล (model) ที่ใช้ ซึ่งโมเดลเป็นกลุ่มของข้อมูลที่เลือกไว้และกฎที่ใช้ในการประมวลผลข้อมูลเหล่านั้นเพื่อที่จะหาโค้ดที่สามารถแทนข้อมูลเหล่านั้นได้ โปรแกรมจะใช้โมเดลในการคำนวณหาความน่าจะเป็นของการซ้ำซ้อนของข้อมูลแต่ละตัวเพื่อทำการสร้างโค้ดสำหรับข้อมูลนั้น ๆ ในด้านการบีบอัดข้อมูลโดยส่วนใหญ่เรามักจะใช้คำว่า coding เพื่อหมายถึงขบวนการบีบอัดข้อมูลทั้งขบวนการแทนที่จะหมายถึงส่วนหนึ่งของขบวนการ เช่น เราอาจจะได้ยินคำว่า Huffman coding ถูกใช้ในการอธิบายเทคนิคการบีบอัดข้อมูล แต่ในความเป็นจริงมันเป็นเพียงวิธีการเข้ารหัสข้อมูลที่ถูกนำมาใช้ร่วมกับโมเดลในการบีบอัดข้อมูลเท่านั้น ดังนั้นเราจึงสามารถนิยามการบีบอัดข้อมูลได้ดังนี้

Data Compression = Modeling + Coding

ถ้าเราดูตัวอย่างจากอัลกอริทึมของ Huffman coding ขบวนการบีบอัดข้อมูลสามารถแบ่งออกได้เป็นขั้นตอนย่อย ๆ ได้ดังในรูปที่ 1

Coding

ในการบีบอัดข้อมูลเราต้องการสร้างโค้ดสำหรับแต่ละตัวอักษรโดยใช้จำนวนบิตตามความเป็นจริง เช่น ตัวอักษร e ใช้ 4 บิต หรือตัวอักษร a ใช้ 6 บิต เป็นต้น แต่ในความเป็นจริงถ้าหากเราเข้ารหัสตัวอักษรโดยใช้ ASCII เราก็ไม่สามารถที่จะเข้ารหัสทุก ๆ ตัวอักษรได้อย่างกะทัดรัดที่สุด ซึ่งในการแก้ไขปัญหาเหล่านี้จึงเป็นที่นิยมในการใช้เทคนิค Shannon-Fano coding และ Huffman coding ในการสร้างโค้ดที่มีความยาวแปรเปลี่ยนได้ โดยที่จำนวนบิตของโค้ดแต่ละตัวก็ขึ้นกับความน่าจะเป็นของการซ้ำซ้อนของข้อมูลในกลุ่มนั้น

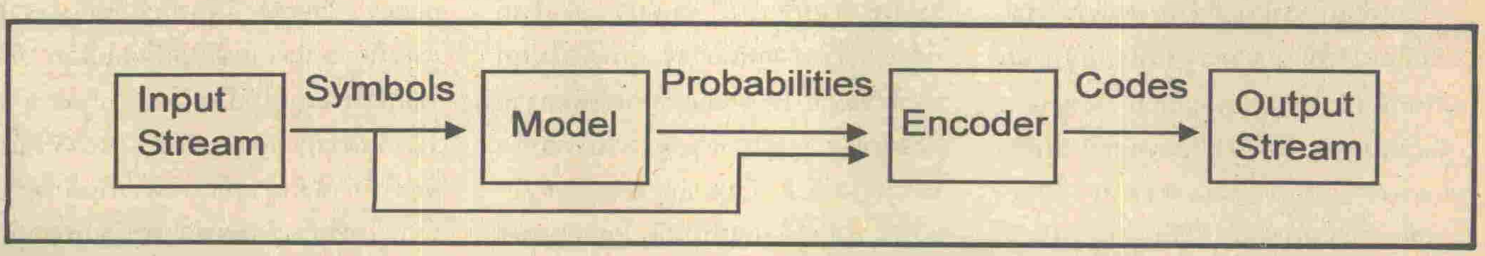
Modeling

ถ้าหากเรามองการบีบอัดข้อมูลเป็นรถจักรยานยนต์ coding ก็จะเป็นส่วนวงล้อและ modeling ก็จะเป็นส่วนเครื่องยนต์ ดังนั้นถ้าหากเราไม่เลือกโมเดลที่ดีพอในการป้อนข้อมูลให้กับส่วนเข้ารหัส อาจจะทำให้การบีบอัดข้อมูลมีประสิทธิผลที่ต่ำก็ได้ ซึ่งการบีบอัดข้อมูลในตระกูล lossless data compression โดยทั่วไปจะมีการเลือกใช้เพียง 2 โมเดลคือ statistical และ dictionary-based โดยที่โมเดล statistical จะอ่านข้อมูลและทำการเข้ารหัสแต่ละตัวอักษรในเวลาเดียวกัน ซึ่งโค้ดที่ได้จะขึ้นกับความน่าจะเป็นของการซ้ำซ้อนของแต่ละตัวอักษร ส่วนโมเดล dictionary-based จะใช้โค้ดตัวหนึ่งที่มีในคำศัพท์ (dictionary) ในการแทนที่สตริงชุดหนึ่ง ๆ โดยถ้าหากพบสตริงดังกล่าวผลลัพธ์ที่ได้ก็จะเป็นค่าอินเด็กซ์หรือค่าพอยน์เตอร์สำหรับตำแหน่งที่พบในคำศัพท์แทนที่จะเป็นโค้ดสำหรับสตริงนั้น ๆ

Ziv and Lempel

ก่อนที่จะจบบทความตอนนี้ก็จะขอกล่าวถึงอัลกอริทึมที่เป็นที่นิยมในปัจจุบันนั่นก็คือ LZW โดยในปี ค.ศ. 1977-1978 Jacob Ziv และ Abraham Lempel ได้อธิบายถึงวิธีการบีบอัดข้อมูลที่ใช้โมเดล adaptive dictionary ซึ่งเป็นอัลกอริทึมใหม่ที่ช่วยให้อัตราส่วนการบีบอัดข้อมูล (compression ratio) สูงขึ้น

LZ77 เป็นอัลกอริทึมแรกที่อธิบายโดย Ziv และ Lempel ซึ่งอธิบายไว้ว่าคำศัพท์ประกอบด้วยสตริงทั้งหมดในหน้าต่างของข้อมูลที่อ่านในครั้งผ่านมา เช่น ถ้าโปรแกรมบีบอัดข้อมูลใช้หน้าต่างขนาด 4K เป็นคำศัพท์ เมื่อมีการอ่านสายข้อมูลใหม่เข้ามา อัลกอริทึมก็จะมองหาสตริงที่ตรงกันในหน้าต่างขนาด 4K ที่ผ่านมา และสตริงส่วนที่พบก็จะถูกเข้ารหัสเป็นค่าพอยน์เตอร์และใช้เป็นผลลัพธ์แทนการเข้ารหัส ซึ่งทำให้เกิดผลดีอย่าง



รูปที่ 1 แสดงขบวนการบีบอัดข้อมูลโดยใช้ Statistical Model และ Huffman Coding

มากเพราะอัลกอริทึมสามารถดูแลง่าย ไม่ซับซ้อน การเข้ารหัสข้อมูลทำได้ง่าย ทำงานได้รวดเร็ว ซึ่งโปรแกรมบีบอัดข้อมูลที่เป็นที่นิยมในปัจจุบันเช่น PKZIP หรือ LHARC ล้วนแต่ใช้อัลกอริทึมที่แยกย่อยไปจาก LZ77 ทั้งนั้น

และต่อมาอัลกอริทึม LZ77 และ LZ78 (ไม่กล่าวถึงรายละเอียดในที่นี้) ก็ได้รับการปรับปรุงโดย Terry Welch ซึ่งเป็นอัลกอริทึมที่ได้รับการยอมรับอย่างสูงในปัจจุบัน และได้ถูกใช้โดยโปรแกรมบีบอัดข้อมูลที่ทำงานบนระบบปฏิบัติการยูนิกซ์ COMPRESS

Lossy Compression

การบีบอัดข้อมูลแบบ lossy เป็นการบีบอัดที่ยอมให้มีการสูญหายของข้อมูลได้บางส่วน ซึ่งโดยทั่วไปมักจะถูกนำไปใช้กับการบีบอัดข้อมูลอะนาล็อกที่ถูกจัดเก็บในรูปแบบดิจิตอล ไฟล์กราฟิกอิมเมจ และไฟล์เก็บเสียง (sound) การบีบอัดข้อมูลประเภทนี้มีการทำงาน 2 เที่ยว โดยที่เที่ยวแรกจะกระทำกับข้อมูลโดยใช้ฟังก์ชันการประมวลผลสัญญาณระดับความถี่สูง ซึ่งในขั้นตอนนี้จะมีการแปลงให้สัญญาณเหล่านี้อยู่ในค่าความถี่โดเมน โดยการใช้อัลกอริทึม FFT-like ซึ่งในส่วนนี้เองจะทำให้เกิดการสูญหายของสัญญาณบางส่วน สำหรับเที่ยวสุดท้ายค่าความถี่แต่ละจุดก็จะถูกบีบอัดโดยใช้เทคนิคตามที่มีในการบีบอัดข้อมูลแบบ lossless

ส่วนการนำไปใช้กับงานด้านกราฟิกอิมเมจก็มี 2 มาตรฐานที่ยอมรับ คือ JPEG (Joint Photographic Experts Group) และ MPEG (Moving Pictures Expert Group) ซึ่งมาตรฐาน JPEG จะทำงานโดยใช้อัลกอริทึม DCT (Discrete Cosine Transform) ในการแปลง

กราฟิกอิมเมจให้เป็นค่าความถี่โดเมน ซึ่งตามมาตรฐานของมันมีการกำหนดค่าแฟกเตอร์คุณภาพ (quality factor) โดยมีค่าตั้งแต่ 0 ถึง 100 และยอมให้ตัวบีบอัดข้อมูลสามารถกำหนดค่าแฟกเตอร์ที่จะใช้ได้

คำที่ควรรู้สำหรับด้านการบีบอัดข้อมูล

เพื่อให้เกิดความเข้าใจที่ตรงกัน ผมจึงขอกล่าวถึงความหมายของคำศัพท์บางคำที่เป็นศัพท์ที่มีความหมายเฉพาะหรือถูกใช้เฉพาะด้านการบีบอัดข้อมูล เพื่อให้เกิดความเข้าใจตรงกันเมื่อมีการนำไปอ้างถึงในตอนต่อ ๆ ไป ดังนี้

Archive

หมายถึงไฟล์ที่ประกอบด้วยหลาย ๆ ไฟล์เข้าด้วยกันเป็นไฟล์เดียว ซึ่งไฟล์ที่ถูกรวมเข้าด้วยกันอาจจะถูกหรือไม่ถูกบีบอัดข้อมูลก็ได้ ซึ่งมันจะถูกใช้เพื่อความสะดวกในการขนถ่ายข้อมูลหรือไฟล์ ซึ่งโปรแกรมบีบอัดข้อมูลเช่น ARC, PKZIP หรือ LHarc ล้วนแต่บีบอัดข้อมูลของแต่ละไฟล์ก่อนที่จะสร้างไฟล์ archive ขึ้นมา

CCITT

CCITT ย่อมาจากคำว่า International Telegraph and Telephone Consultative Committee ซึ่งเป็นชื่อขององค์การควบคุมมาตรฐานทางด้านวิธีการบีบอัดข้อมูลหรือขนถ่ายข้อมูลในปัจจุบัน ซึ่งมีอยู่หลาย ๆ เทคนิค เช่น PCM และ ADPCM การส่งข้อความโดยใช้แฟกซ์

(En) Coding

ขบวนการเข้ารหัส (coding) เป็นขบวนการที่เครื่องคอมพิวเตอร์ทำการแปลงสัญลักษณ์ต่าง ๆ ให้สามารถถูกจัดเก็บในรูปแบบรหัสต่าง ๆ ได้ เช่น ASCII หรือ EBCDIC เป็นต้น

Compression ratio

ค่าอัตราส่วนการบีบอัดได้ถูกนำไปใช้ในการอธิบายความแตกต่างของขนาดของไฟล์ต้นฉบับและไฟล์ที่ได้จากการบีบอัดข้อมูล ซึ่งการคำนวณค่าดังกล่าวนี้ก็มีอยู่ด้วยกันหลาย ๆ วิธี ซึ่งวิธีหนึ่งที่ถูกใช้กันอย่างแพร่หลายก็คือ การบอกค่าเป็นอัตราส่วนระหว่างอินพุตและเอาต์พุต เช่น "4:1" ส่วนอีกวิธีที่ได้รับการใช้กันอย่างมากเช่นกันก็คือการบอกเป็นเปอร์เซ็นต์ ซึ่งมีค่าตั้งแต่ 0 ถึง 100 เปอร์เซ็นต์

CRC

ค่าตัวเลข CRC (Cyclical Redundancy Check) ได้ถูกสร้างขึ้นมาจากสูตรที่คำนวณจากข้อมูลทีละบล็อก ซึ่งโดยทั่วไปจะถูกใช้เป็นค่าของ checksum โดยที่สูตรการคำนวณค่า CRC ที่ดีนั้นต้องให้ค่าดังกล่าวที่แตกต่างกันสำหรับเงื่อนไขที่เกิดข้อผิดพลาดที่แตกต่างกันอย่างหลากหลาย

JPEG

ย่อมาจากคำว่า Joint Photographic Experts Group โดยสาเหตุที่ใช้คำว่า Joint เริ่มต้นก็เพราะว่าคณะกรรมการชุดนี้ได้ถูกแทรกแซงจากองค์กร CCITT และ ISO นั่นเอง ซึ่งคณะกรรมการชุดนี้ได้ทำงานด้านมาตรฐานการบีบอัดข้อมูลประเภทกราฟิกอิมเมจ ซึ่งประเภทของการบีบอัดที่ใช้ก็เป็น lossy compression

Order

คำว่า "ลำดับของโมเดล (order of model)" หมายถึงจำนวนของสัญลักษณ์ที่ผ่านมาที่ถูกนำมาใช้พิจารณาการเข้ารหัสของข้อมูลปัจจุบัน เช่น โมเดล order-0 จะไม่มีการนำสัญลักษณ์ที่ผ่านมาเข้ามาพิจารณาวิธีการเข้ารหัสข้อมูลปัจจุบัน

MPEG

ย่อมาจากคำว่า Moving Pictures Experts Group ซึ่งถูกควบคุมโดยองค์การ ISO ซึ่งทำงานด้านคุณภาพการส่งข้อมูลดิจิตอลทั้งด้านภาพวิดีโอเคลื่อนที่และเสียงผ่านทางการสื่อสาร ซึ่งจุดเด่นของกลุ่ม MPEG ก็คือ สามารถส่งภาพและเสียงสเตอริโอที่มีคุณภาพสูงผ่านทางเซลเนล 1.5 Mbps

เมื่อเราเริ่มรู้จักกับการบีบอัดข้อมูลพอสมควรแล้ว ผมก็ขอยกตัวอย่างวิธีการเข้ารหัสข้อมูลที่มีประสิทธิภาพ วิธีแรกที่เป็นที่รู้จักกันดีก็คือ Shannon-Fano coding ซึ่งได้รับการพัฒนาจาก Claude Shannon แห่ง Bell Labs และ R.M. Fano แห่งมหาวิทยาลัย M.I.T. ซึ่งบุคคลทั้งสองได้พัฒนาอัลกอริทึมดังกล่าวขึ้นมาในเวลาที่ไล่เลี่ยกันมาก หลักการทำงานของอัลกอริทึมนี้อาศัยพื้นฐานของความน่าจะเป็นของแต่ละตัวอักษร และสิ่งที่ได้จากวิธีการเข้ารหัสวิธีนี้มีดังนี้

- โค้ดที่แตกต่างกันจะมีการจัดลำดับบิตที่แตกต่างกัน
- โค้ดสำหรับสัญลักษณ์ที่มีความซ้ำซ้อนต่ำกว่าจะมีจำนวนบิตมากกว่า และโค้ดสำหรับสัญลักษณ์ที่ความซ้ำซ้อนมากกว่าจะมีจำนวนบิตน้อยกว่า
- เนื่องจากโค้ดมีจำนวนบิตที่ไม่เท่ากัน ดังนั้นจึงสามารถถอดรหัสกลับได้เหมือนเดิมทุกประการ

เนื่องจากโค้ดแต่ละตัวมีจำนวนบิตที่ไม่เท่ากัน ดังนั้นในการถอดรหัสโค้ดเหล่านี้เราจึงนิยมใช้วิธีการเรียงโค้ดโดยใช้ binary tree ซึ่งตัวอย่างในรูปที่ 2 แสดงการถอดรหัส binary tree ที่ถูกนำมาใช้กับ Shannon-Fano coding ซึ่งการเรียงลำดับบิตของโค้ดแต่ละตัวจะเริ่มต้นจากรากลงไปตามกิ่งสาขาทั้งซ้ายและขวา และโค้ดแต่ละตัวก็จะสิ้นสุดที่ปลายของแต่ละสาขา

จากรูปที่ 2 เราสามารถที่จะเข้าหรือถอดรหัสสำหรับตัวอักษรแต่ละตัวได้โดยง่าย โดยการเดินตามกิ่งสาขาของต้นไม้ ซึ่งจะได้โค้ดดังตัวอย่างข้างล่าง

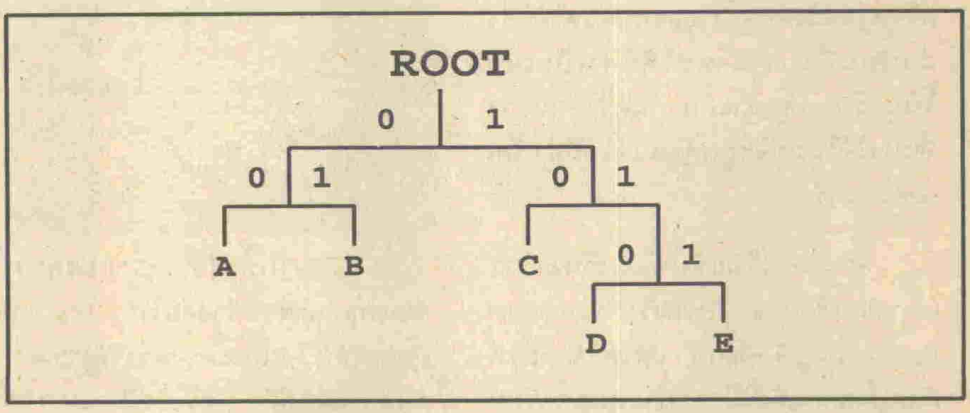
ตัวอักษร	โค้ด
A	00
B	01
C	10
D	110
E	111

อัลกอริทึมของ Shannon-Fano

แผนภูมิต้นไม้ของ Shannon-Fano สามารถสร้างขึ้นมาได้โดยใช้อัลกอริทึมพิเศษที่ถูกออกแบบมาเพื่อหาตารางของโค้ดที่มีประสิทธิผลในการช่วยลดขนาดของข้อมูล ซึ่งมีรายละเอียดตามขั้นตอนดังนี้

1. จะต้องทำการหาค่าความน่าจะเป็นหรือนับค่าความถี่ของการซ้ำซ้อนของแต่ละตัวอักษร ดังนั้นเราจะทราบถึงค่าความถี่ของการเกิดทุก ๆ แต่ละตัวอักษร

2. จัดเรียงลำดับแต่ละตัวอักษรที่มีตามค่าความถี่ที่ได้ โดยตัวอักษรที่มีค่าความถี่มากที่สุดจะอยู่บนสุดและตัวอักษรที่มีค่าความถี่น้อยที่สุดก็จะอยู่ล่างสุด



รูปที่ 2 แสดงแผนภูมิต้นไม้ของ Shannon-Fano

3. แยกรายการที่จัดเรียงออกเป็น 2 ส่วน โดยให้ค่าความถี่รวมของทั้งสองส่วนมีค่าเท่ากันหรือมีค่าที่ใกล้เคียงกันมากที่สุดเท่าที่จะเป็นไปได้

4. ครึ่งส่วนบนทั้งหมดจะถูกกำหนดให้มีค่าเป็นเลขไบนารี 0 และครึ่งส่วนล่างทั้งหมดจะถูกกำหนดให้มีค่าเป็นเลขไบนารี 1 ซึ่งหมายความว่าโค้ดสำหรับครึ่งส่วนบนทั้งหมดจะถูกเริ่มต้นด้วยเลขไบนารี 0 และเลขในครึ่งส่วนล่างทั้งหมดก็จะถูกเริ่มต้นด้วยเลข 1

5. ทำการซ้ำเดิมตามขั้นตอนที่ 3 โดยการแบ่งครึ่งของทั้งครึ่งส่วนบนและครึ่งส่วนล่างอีกครั้ง และกำหนดให้ครึ่งส่วนบนที่แบ่งได้ใหม่ให้มีค่าเป็นเลขไบนารี 0 และเช่นเดียวกันครึ่งส่วนล่างที่แบ่งได้ใหม่ก็จะถูกกำหนดให้มีค่าเป็นเลขไบนารี 1

จากอัลกอริทึมข้างต้นจะเห็นว่าการเข้ารหัสหรือถอดรหัสของวิธี Shannon-Fano ไม่ใช่สิ่งที่ยาก แต่โพรซีเดอร์ที่เขียนขึ้นมาจะต้องมีการทำงานแบบเรียกตัวเอง (Recursive function) ได้

ตัวอย่างการเข้ารหัสด้วยวิธี Shannon-Fano

ตัวอย่างต่อไปนี้เป็นการแสดงการเข้ารหัสข้อมูลด้วยวิธี Shannon-Fano โดยสุดท้ายก็จะสามารถเขียนโค้ดที่ได้ในรูปแบบแผนภูมิต้นไม้ดังในรูปที่ 2 สำหรับในขั้นตอนแรกคือการเรียงลำดับข้อมูลตามค่าความถี่มากไปยังค่าความถี่น้อย และเพื่อเป็นการประหยัดเวลาก็สมมติว่าข้อมูลได้ถูกนับค่าความถี่และเรียงลำดับดังนี้

ข้อมูล	ค่าความถี่
A	15
B	7
C	6
D	6
E	5

จากนั้นก็ทำการแบ่งข้อมูลที่จัดเรียงลำดับออกเป็น 2 ส่วนโดยให้มีค่ารวมของความถี่ที่ใกล้เคียงกันมากที่สุด ซึ่งจะเห็นว่าครึ่งบนจะมีข้อมูลของ A และ B ซึ่งรวมกันได้ 22 ตัวอักษร ดังนั้นครึ่งส่วนบนจึงเริ่มต้นด้วยเลขไบนารี 0 และครึ่งล่างจะประกอบด้วยตัวอักษร C, D และ E ซึ่งรวมค่าความถี่ได้เป็น 17 และจำนำด้วยเลขไบนารี 1 ดังนี้

ข้อมูล	ค่าความถี่	
A	15	0
B	7	0
----------	----------	แบ่งครั้งที่ 1
C	6	1
D	6	1
E	5	1

หลังจากนั้นก็ทำการแบ่งครึ่งทั้งส่วนบนและส่วนล่างต่อไป โดยเราจะเริ่มจากครึ่งบนก็จะแบ่งครึ่งได้ง่ายเพราะเหลือข้อมูลเพียง 2 ตัว ดังนั้นจึงสามารถแบ่งครึ่งระหว่าง A และ B ได้เลย และ A ก็จะอยู่บนกิ่งสาขารหัส 00 และ ก็จะอยู่บนกิ่งสาขารหัส 01 จากนั้น แบ่งครึ่งล่าง แต่เนื่องจากในครึ่งส่วนล่า มีข้อมูลอยู่ 3 ตัว ดังนั้นก็จะต้องทำกา แบ่งครึ่ง 2 อีก ซึ่งสุดท้ายก็จะได้โค้ สำหรับแต่ละตัวอักษรที่มีจำนวนบิตทั้ง และ 3 บิต ดังนี้

ข้อมูล	ค่าความถี่			
A	15	0	0	
----------	----------	แบ่งครั้งที่		
B	7	0	1	
----------	----------	แบ่งครั้งที่		
C	6	1	0	
----------	----------	แบ่งครั้งที่		
D	6	1	1	0
----------	----------	แบ่งครั้งที่		
E	5	1	1	1

จะเห็นได้ว่าข้อมูลที่มีค่าความ มากกว่าจะมีจำนวนบิตที่น้อยกว่า แล เมื่อเปรียบเทียบกับเมื่อเราใช้ตัวอักษ รหัส ASCII ขนาด 8 บิตจะเห็นว่ ข้อมูลทั้งหมดมีความยาวเท่ากับ 8x39 หรื 312 บิต แต่โค้ดที่ได้จากวิธี Shannon Fano จะทำให้ข้อมูลทั้งหมดมีความยา เพียง 30+14+12+18+15 หรือ 89 บิตเท่ นั้น แต่ในการจัดเก็บข้อมูลโดยใช้โค้ดที่ ความยาวไม่เท่ากันนี้ไม่สามารถกระทำ ได้ในระบบคอมพิวเตอร์ ดังนั้นเราจึงนำ โค้ดที่มีจำนวนบิตที่ไม่เท่ากันนี้มาเรีย ต่อกันและนำมาเข้ารหัสกลับเป็นรหั ASCII อีกครั้ง แต่เราจะต้องสร้างโพร ซีเดอร์ที่สามารถถอดรหัสโค้ดเดิมที่ ความยาวไม่เท่ากันกลับคืนได้เช่นกัน

โปรแกรม BITIO.C ในลิสติงที่ เป็นโปรแกรมที่ประกอบด้วยโพรซีเดอ ที่สามารถนำมาใช้ร่วมกันสำหรับกา บีบอัดข้อมูลโดยวิธีต่าง ๆ ดังที่จะกล่าวใ ตอนต่อ ๆ ไป ดังนั้นเพื่อเป็นการประหยั เวลาและเนื้อที่หน้ากระดาษจึงได้แย

ตัวอักษร	ค่าความถี่	จำนวนบิต	จำนวนบิตทั้งหมด
A	15	2	15×2 = 30 บิต
B	7	2	7×2 = 14 บิต
C	6	2	6×2 = 12 บิต
D	6	3	6×3 = 18 บิต
E	5	3	5×3 = 15 บิต

ตารางที่ 1

โพรซีเดอร์ส่วนที่สามารถใช้ร่วมกันได้ออกมาเป็นโปรแกรมแยกต่างหาก และถ้าต้องการรันโปรแกรมก็ให้ทำการคอมพายล์โดยรวมเอาโปรแกรมนี้เข้าไว้ด้วยกัน โปรแกรมประเภทบีบอัดข้อมูลจะมีการกระทำกับจำนวนบิตที่ไม่แน่นอน ซึ่งการที่จะเขียนหรืออ่านข้อมูลทีละบิตในแต่ละครั้งก็ไม่สามารถกระทำได้โดยใช้รูทีนในไลบรารีมาตรฐานของภาษาซี ดังนั้นในส่วนนี้จึงต้องมีการปรับปรุงโค้ดส่วนนี้เสียใหม่เพื่อให้เหมาะสมกับรูทีนมาตรฐานที่มีอยู่ ซึ่งแต่ละรูทีนในไฟล์ BITIO.C มีหน้าที่ดังนี้

OpenInputBitFile( )

เปิดไฟล์สำหรับอ่านข้อมูลเข้ามาโดยใช้คำสั่ง fopen( ) และคืนกลับพอยน์เตอร์โครงสร้าง FILE ในตัวแปรโครงสร้าง BIT__FILE

OpenOutputBitFile( )

เปิดไฟล์สำหรับอ่านข้อมูลส่งออกไปโดยใช้คำสั่ง fopen( ) และคืนกลับพอยน์เตอร์โครงสร้าง FILE ในตัวแปรโครงสร้าง BIT__FILE

CloseInputBitFile( )

ปิดไฟล์สำหรับอ่านข้อมูลเข้ามาโดยใช้คำสั่ง fclose( ) และคืนกลับหน่วยความจำที่จองที่สำหรับตัวแปรโครงสร้าง BIT__FILE เอาไว้

CloseOutputBitFile( )

ปิดไฟล์สำหรับอ่านข้อมูลส่งออกไปโดยใช้คำสั่ง fclose( ) และคืนกลับหน่วยความจำที่จองที่สำหรับตัวแปรโครงสร้าง BIT__FILE เอาไว้

OutputBit( )

เขียนข้อมูลครั้งละ 1 บิตลงในไฟล์ที่กำหนด

OutputBits( )

เขียนข้อมูลครั้งละหลาย ๆ บิตลงในไฟล์ที่กำหนด (ขึ้นกับขนาดของตัวแปร unsigned int)

InputBit( )

อ่านข้อมูลครั้งละ 1 บิตจากไฟล์ที่กำหนด

InputBits( )

อ่านข้อมูลครั้งละหลาย ๆ บิตจากไฟล์ที่กำหนด (ขึ้นกับขนาดของตัวแปร unsigned int)

fatal_error( )

รูทีนนี้จะถูกเรียกเมื่อเกิดข้อผิดพลาดประเภท unrecoverable ในโปรแกรม

สำหรับในตอนต่อไปจะกล่าวถึงวิธีการเขียนโปรแกรมการบีบอัดข้อมูลด้วยวิธี Huffman coding ซึ่งเป็นวิธีการบีบอัดข้อมูลที่คล้ายกับ Shannon-Fano coding แต่จะมีลำดับการเรียงโค้ดที่ไม่เหมือนกัน สำหรับอัลกอริทึมแสดงการทำงานของวิธี Huffman Coding ก็ขอให้ผู้อ่านติดตามจากวารสารไมโครคอมพิวเตอร์ในฉบับที่ 110 ครับ

ลิสติงที่ 1
โปรแกรม BITIO.H

```
/*
 * BITIO.H
 */

#ifndef _BITIO_H
#define _BITIO_H

#include <stdio.h>

typedef struct bit_file {
    FILE *file;
    unsigned char mask;
    int rack;
    int pacifier_counter;
} BIT_FILE;

BIT_FILE      *OpenInputBitFile();
BIT_FILE      *OpenOutputBitFile();
void           OutputBit();
void           OutputBits();
int            InputBit();
unsigned long  InputBits();
void           CloseInputBitFile();
void           CloseOutputBitFile();
void           fatal_error();

#endif  /* _BITIO_H */
```

ลิสติงที่ 2 โปรแกรม BITIO.C

```
/*
 * BITIO.C   This utility file contains all of the routines needed to
 *           impement bit oriented routines under C. It needs to be linked
 *           with every program.
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "bitio.h"

#define PACIFIER_COUNT 2047

BIT_FILE *OpenOutputBitFile( name )
char *name;
{
    BIT_FILE *bit_file;

    bit_file = (BIT_FILE *) calloc( 1, sizeof( BIT_FILE ) );
    if ( bit_file == NULL )
        return( bit_file );
    bit_file->file = fopen( name, "wb" );
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    bit_file->pacifier_counter = 0;
    return( bit_file );
}

BIT_FILE *OpenInputBitFile( name )
char *name;
{
    BIT_FILE *bit_file;

    bit_file = (BIT_FILE *) calloc( 1, sizeof( BIT_FILE ) );
    if ( bit_file == NULL )
        return( bit_file );
    bit_file->file = fopen( name, "rb" );
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    bit_file->pacifier_counter = 0;
    return( bit_file );
}

void CloseOutputBitFile( bit_file )
BIT_FILE *bit_file;
{
    if ( bit_file->mask != 0x80 )
        if ( putc( bit_file->rack, bit_file->file ) != bit_file->rack )
            fatal_error( "Fatal error in CloseBitFile!\n" );
    fclose( bit_file->file );
    free( (char *) bit_file );
}

void CloseInputBitFile( bit_file )
BIT_FILE *bit_file;
{
    fclose( bit_file->file );
    free( (char *) bit_file );
}

void OutputBit( bit_file, bit )
BIT_FILE *bit_file;
int bit;
{
    if ( bit )
        bit_file->rack |= bit_file->mask;
    bit_file->mask >>= 1;
    if ( bit_file->mask == 0 ) {
        if ( putc( bit_file->rack, bit_file->file ) != bit_file->rack )
            fatal_error( "Fatal error in OutputBit!\n" );
        else
        if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
            putc( '.', stdout );
        bit_file->rack = 0;
        bit_file->mask = 0x80;
    }
}

void OutputBits( bit_file, code, count )
BIT_FILE *bit_file;
unsigned long code;
int count;
{
    unsigned long mask;
```

ลิสติงที่ 2 (ต่อ)

```
    mask = 1L << ( count - 1 );
    while ( mask != 0) {
        if ( mask & code )
            bit_file->rack |= bit_file->mask;
        bit_file->mask >>= 1;
        if ( bit_file->mask == 0 ) {
            if ( putc( bit_file->rack, bit_file->file ) != bit_file->rack )
                fatal_error( "Fatal error in OutputBit!\n" );
        else if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
                putc( '.', stdout );
            bit_file->rack = 0;
            bit_file->mask = 0x80;
        }
        mask >>= 1;
    }
}

int InputBit( bit_file )
BIT_FILE *bit_file;
{
    int value;

    if ( bit_file->mask == 0x80 ) {
        bit_file->rack = getc( bit_file->file );
        if ( bit_file->rack == EOF )
            fatal_error( "Fatal error in InputBit!\n" );
    if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
            putc( '.', stdout );
    }
    value = bit_file->rack & bit_file->mask;
    bit_file->mask >>= 1;
    if ( bit_file->mask == 0 )
       bit_file->mask = 0x80;
    return( value ? 1 : 0 );
}

unsigned long InputBits( bit_file, bit_count )
BIT_FILE *bit_file;
int bit_count;
{
    unsigned long mask;
    unsigned long return_value;

    mask = 1L << ( bit_count - 1 );
    return_value = 0;
    while ( mask != 0) {
        if ( bit_file->mask == 0x80 ) {
            bit_file->rack = getc( bit_file->file );
            if ( bit_file->rack == EOF )
                fatal_error( "Fatal error in InputBit!\n" );
        if ( ( bit_file->pacifier_counter++ & PACIFIER_COUNT ) == 0 )
            putc( '.', stdout );
        }
        if ( bit_file->rack & bit_file->mask )
            return_value |= mask;
        mask >>= 1;
        bit_file->mask >>= 1;
        if ( bit_file->mask == 0 )
            bit_file->mask = 0x80;
    }
    return( return_value );
}

void fatal_error( fmt )
char *fmt;
{
    va_list argptr;

    va_start( argptr, fmt );
    printf( "Fatal error: " );
    vprintf( fmt, argptr );
    va_end( argptr );
    exit( -1 );
}
```